Big Data intelligence Homework 3

Christoffer Brevik

December 16, 2024

Introduction

This report explains my findings and the results of my code regarding the usage of different machine learning architectures for the classification of letters and symbols in different languages. I utilized the `omniglot` dataset provided in the homework, which includes characters from 50 distinct written languages. The task involves comparing two approaches to classify 50 randomly selected characters from this dataset, using a training set of 15 images per character and a test set of 5 images per character.

Note about the programs

My code uses the *Pandas* library to import and preprocess data, for matrix operations it uses the *numpy* library and for plotting it uses *pyplot*. The original code was developed in a Jupyter Notebook and subsequently exported to a .py file, as required by the homework.

I used the keras library to develop the neural networks. With *Conv2D* and *Dense* layers as the fundamentals for the two models.

0.1 Report Structure

I will in the report discuss: *The loading and preprocessing of data*, *My fully connected model* and my *convoluted model*, before discussing the results of my *Experiments on the models*, before drawing a conclusion on the two models for image classification

1 Loading and preprocessing data

To import the data correctly, I used the provided *utils.py* file. Here I replaced the `misc` library with `PIL` as the misc library was depricated. Apart from this and a few other small changes, the functions remain mostly unchanged. I will now briefly explain the functions that my program uses:

LoadData

The `LoadData` function is responsible for loading the Omniglot dataset, splitting it into training and testing subsets, and optionally reshaping or flattening the data. This function ensures reproducibility and consistency by setting a random seed and performs the following steps:

1. **Set Parameters and Validate Inputs:** The function accepts parameters such as the number of classes, training/testing samples per class, random seed, and whether to flatten the images. It also validates the constraints on the number of classes and samples.
2. **Locate Dataset Folders:** The dataset is organized into character folders grouped by families. The function builds a list of paths to these character folders, shuffled to randomize the data.
3. **Read and Process Images:** Using the `get_images` function, the program reads image paths for the specified number of samples per class. The `image_file_to_array` function then converts these image files into normalized numpy arrays with pixel values between 0 and 1.
4. **Split Data into Training and Testing:** The program splits the dataset into training and testing subsets based on the specified number of samples per class. The images and labels are shuffled for both subsets using the `pair_shuffle` function.
5. **Optional Flattening:** If the `flatten` parameter is set to true, the function reshapes the images into one-dimensional arrays for compatibility with certain machine learning models.

The function returns four outputs: training images, training labels, testing images, and testing labels.

Below are a short description of the helper functions used by `LoadData`:

get_images

The `get_images` function retrieves a specified number of images from each character folder and pairs them with corresponding labels. It supports shuffling the image-label pairs to ensure randomness. This function works as follows:

1. **Sample Images:** If a specific number of samples is required, it uses the `random.sample` function to select images from the folder.
2. **Pair Images and Labels:** Each selected image is paired with the label corresponding to its character class.
3. **Shuffle Pairs:** The pairs are shuffled to introduce randomness into the dataset.

The function returns a list of tuples where each tuple contains a label and the corresponding image path.

image_file_to_array

The `image_file_to_array` function converts an image file into a normalized numpy array. It performs the following operations:

1. **Open Image:** The image file is opened and converted to grayscale using the `PIL` library.
2. **Normalize Pixels:** The pixel values are scaled to the range [0, 1] by dividing by 255 and inverting the values to match the desired format.
3. **Return Array:** The processed image is returned as a numpy array.

This function ensures compatibility with the downstream machine learning pipeline.

array_to_image_file

The `array_to_image_file` function reverses the transformation performed by `image_file_to_array`, converting a numpy array back into an image file. The key steps include:

1. **Reshape Array:** The array is reshaped into the dimensions of the original image.
2. **Reverse Normalization:** The inversion and scaling applied earlier are undone.
3. **Save Image:** The resulting image is saved to the specified file path using `PIL`.

This function is useful for visualizing and validating the preprocessed data.

pair_shuffle

The `pair_shuffle` function ensures that the correspondence between images and labels is maintained while shuffling. It takes two arrays (images and labels) and returns them shuffled consistently using the same permutation.

1. **Generate Permutation:** A random permutation of indices is created.
2. **Apply Permutation:** Both arrays are shuffled using this permutation.

This function helps in creating randomized datasets while preserving the integrity of image-label mappings.

2 Fully Connected Neural Network

This section discusses my implementation of a Fully Connected Neural Network (FCNN) for the classification assignment, including the architecture, parameter experimentation, and results.

2.1 Architecture

The assignment required atleast one hidden layer, so I tested with both one, two and three. Betweeen each *Dense* layer, I added *Dropoput*. The architecture of my FCNN models are implemented using Keras with TensorFlow as the backend, and consists of the following components:

- **Input Layer:** The input shape is a flattened 28x28 image (784 features).
- **Hidden Layers:**
  - The first hidden layer has 512 units with ReLU activation and a dropout rate of 0.3 for regularization.
  - The second hidden layer has 256 units with ReLU activation, followed by another dropout layer with a 0.3 rate.
  - The third hidden layer has 128 units with ReLU activation.
- **Output Layer:** The output layer has as many units as the number of classes (50 in this case) and uses a softmax activation function to output class probabilities.

The model is built using the `Sequential` API and compiled with the Adam optimizer, Sparse Categorical Crossentropy as the loss function, and Sparse Categorical Accuracy as the evaluation metric.

2.2 3D Visualization of Results

The results of the hyperparameter tuning experiments were visualized using a 3D scatter plot. As the differences between the learning rates were not that noticable, I will still display one of these below for reference, where:

- The X-axis represents learning rates.
- The Y-axis represents the number of epochs.
- The Z-axis and color gradient represent validation accuracy.

I couldn't find a pattern or convergence toward a given learning rate. Therefore this function didn't help me much during experimentation.

2.3 Results

The FCNN achieved, what I would call, a sub-par accuracy in classifying the Omniglot characters, with the best-performing model achieving a validation accuracy of *0.548*. I believe there still might be improvemenets that could make this model perform closer to the convoluted model, which will be explained in the next chapters.

3 Convolutional Neural Network

This section discusses my implementation of a Convolutional Neural Network (CNN) for the classification assignment, including the architecture, experimentation with hyperparameters, and results.

3.1 Architecture

The CNN architecture was inspired by the example provided in the assignment. It incorporates two convolutional blocks followed by two fully connected layers. Each block combines convolutional, batch normalization, ReLU activation, and pooling layers to extract hierarchical features. The architecture is implemented using Keras with TensorFlow as the backend and includes the following components:

- **Input Layer:** The input is an image of shape $28 \times 28 \times 1$, representing grayscale images.
- **Convolutional Layers:**
  - **First Block:**
    - Two convolutional layers with 32 filters each and a kernel size of 3×3.
    - Batch normalization applied after each convolution for faster convergence and improved generalization.
    - ReLU activation to introduce non-linearity.
    - Max pooling with a 2×2 pool size to reduce spatial dimensions.
  - **Second Block:**
    - Two convolutional layers with 64 or 32 filters each and a kernel size of 3×3.
    - Batch normalization and ReLU activation similar to the first block.
    - Max pooling to further reduce dimensions.
- **Fully Connected Layers:**
  - A flattening layer transforms the output of the convolutional layers into a one-dimensional vector.
  - A dense layer with 512 units, batch normalization, and ReLU activation.
  - A dropout layer with a rate of 0.5 for regularization to mitigate overfitting.
  - The final dense layer has as many units as the number of classes (50 in this case) and uses softmax activation for class probability predictions.

This CNN architecture was designed to balance feature extraction and regularization to achieve robust classification.

3.2 3D Visualization of Results

Similar to the FCNN, the CNN results were visualized using a 3D scatter plot to analyze the interplay between learning rate, epochs, and validation accuracy. But as in FCNN, I didn't feel like the visualization helped me find any patterns during experimentation.

3.3 Results

The CNN demonstrated strong performance in classifying the Omniglot character dataset. The highest validation accuracy achieved was *0.924*, which I find very satisfactory. Overall, the CNN provided a robust solution for the classification task, with systematic hyperparameter tuning and regularization techniques ensured optimal performance.

4 Experimentation

As required in the assignment, I wanted to experiment with my models if I added or removed certain aspects of them. I want to focus on tweaking my learning rates and epochs, experiment with changing the split of training- and testdata as well as

4.1 Modifying epochs and learning rate

Here I actually run the algorithms again for each learning rate and epoch combination. This was because this, while taking more time, would give me an insight into randomnes. As you will see this actually gave results, with some runs converging faster than others with fewer epochs on the same learning rate.

I tested with the alpha values:

```
1.0, 1e-1, 1e-2, 1e-3 & 1e-4
```

And had the epoch values:

```
500, 1000 & 2000
```

As there were a lot of combinations, I've summarized my results in the list below.

Fully Connected Model

To optimize the FCNN, I experimented with different learning rates and numbers of epochs. My `train_multiple_models` method facilitated the testing of these hyperparameters by iterating over a grid of learning rates and epoch combinations. With my different models, epochs and learning rates I got the following results:

1 Hidden Layer

Learning Rate (lr)	Epochs	Accuracy	Best Accuracy (per lr)
1.0	500	0.1160	0.1160
	1000	0.0440	
	2000	0.0760	
0.1	500	0.1080	0.1120
	1000	0.1120	
	2000	0.1000	
0.01	500	0.4880	0.5480
	1000	0.5320	
	2000	0.5480	
0.001	500	0.4880	0.5240
	1000	0.4760	
	2000	0.5240	
0.0001	500	0.5200	0.5200
	1000	0.5000	
	2000	0.5200	

2 Hidden Layer

Learning Rate (lr)	Epochs	Accuracy	Best Accuracy (per lr)
1.0	500	0.0440	0.0520

	1000	0.0520	
	2000	0.0320	
0.1	500	0.0360	0.0600
	1000	0.0600	
	2000	0.0200	
0.01	500	0.5240	0.5320
	1000	0.5240	
	2000	0.5320	
0.001	500	0.5120	0.5120
	1000	0.4960	
	2000	0.5040	
0.0001	500	0.4960	0.5280
	1000	0.5280	
	2000	0.4880	

3 Hidden Layer

Learning Rate (lr)	Epochs	Accuracy	Best Accuracy (per lr)
1.0	500	0.0200	0.0200
	1000	0.0200	
	2000	0.0200	
0.1	500	0.0200	0.0200
	1000	0.0200	
	2000	0.0200	
0.01	500	0.1360	0.1520
	1000	0.1520	
	2000	0.0240	
0.001	500	0.4480	0.5160
	1000	0.5040	
	2000	0.5160	
0.0001	500	0.4920	0.4920
	1000	0.4720	
	2000	0.4720	

Best Total Accuracy for Fully Connected Model

From the tables above we see that the best model was the one with 1 hidden layer, a learning rate of 0.01 and running for 2000 epochs. It reached a total accuracy of **0.5480**. As this gave the best results, I will use these paramaters when doing further experimentation

Convoluted Model

To optimize the CNN, I explored various hyperparameters, including learning rates, numbers of epochs, and regularization techniques. The `train_multiple_models` method automated the evaluation of different configurations to identify the best-performing model. With my different epochs and learning rates I got the following results:

Learning Rate (lr)	Epochs	Accuracy	Best Accuracy (per lr)
0.05	500	0.8960	0.9120
	1000	0.9120	

	1500	0.9080	
0.01	500	0.9000	0.9240
	1000	0.9000	
	1500	0.9240	
0.005	500	0.8760	0.9120
	1000	0.8840	
	1500	0.9120	
0.001	500	0.8840	0.8960
	1000	0.8800	
	1500	0.8960	

Best Total Accuracy for Convoluted Model

From the tables above we see that the best model was the one with the learning rate of 0.01 running for 1500 epochs. It reached a total accuracy of **0.9240**. This is way better than the fully connected version. As this gave the best results, I will use these paramaters when doing further experimentation

Discussion

For the FCNN, the experiments revealed that the choice of hyperparameters had a notable impact on convergence and final accuracy. Interestingly, the results were consistent with expectations for smaller learning rates (α), where the model required more epochs to converge but ultimately achieved better accuracy. For instance, with $\alpha = 0.01$, the model attained the best accuracy (0.5480) with 2000 epochs. This suggests that smaller learning rates allowed the model to make more precise updates to the weights, avoiding overshooting the optimal minima. On the larger ln-values the model converged faster but not to a good enough value. The results also demonstrated some randomness in convergence behavior. For example, with $\alpha = 0.001$, the model achieved higher accuracy at 2000 epochs compared to 1000, but the improvement was not consistent across all learning rates. This variability could be attributed to the initialization of weights or the inherent stochastic nature of gradient-based optimization.

The CNN performed significantly better overall compared to the FCNN. The best configuration achieved an accuracy of 0.9240 with $\alpha = 0.01$ and 1500 epochs. This aligns with expectations, an this points to CNNs being better suited for extracting spatial features from image-based datasets. The higher performance at moderate learning rates (e.g., $\alpha = 0.01$) suggests that CNNs were able to converge effectively without overfitting or oscillating. Interestingly, for $\alpha = 0.05$, the accuracy plateaued after 1000 epochs, indicating diminishing returns with additional training time. This suggests that the CNN can achieve optimal results faster than the FCNN, likely due to its architectural advantages. Overall, the experiments confirmed my assumptions about learning rates and epochs. Smaller learning rates and sufficient training time produced the best results, with CNNs outperforming FCNNs due to their ability to handle complex patterns.

4.2 Modifying data split

Here I will test with changing the split between Test/Training Data. As we have a quite small dataset per category I expect quite volile results here. I tested with 4 combinations; 20/28 Split, 50/50 Split,80/20 Split and 90/10 Split. With this I got the following results.

Split	**Training Images**	**Test Images**	**Fully Connected**	**Convolutional**
20/80	4	16	0.5640	0.8799
50/50	10	10	0.4320	0.8059
80/20	16	4	0.5199	0.8399
90/10	18	2	0.5600	0.8700

Discussion

The FCNN showed noticeable fluctuations in accuracy based on the training set size. For instance, the 50/50 split resulted in the lowest accuracy (0.4320), while the 20/80 split (where only 20% of the data was used for training) surprisingly achieved one of the highest accuracies (0.5640). This outcome was unexpected, as smaller training sets typically reduce a model's ability to generalize. The higher accuracy for smaller splits might be explained by overfitting to the limited data, which might have given an artificially inflated performance on the test set.

The CNN exhibited more stable and consistent performance across different splits, with the best results achieved using the 20/80 split (0.8799). However, the slight drop in performance with larger training splits (e.g., 50/50 and 80/20) might indicate overfitting or noise in the additional training data. Overall, while the results for the FCNN were less predictable, the CNN performed as expected, demonstrating its adaptability to various data splits. However, the unexpectedly high accuracy for smaller splits warrants further investigation, possibly through cross-validation.

4.3 Modifying data category size

Here I will test with changing the amount of categories available. Here having a smaller category count should lead to higher accuracy as the models will have fewer options to chose from. Even if a model is totally random, reducing the categories should intuitivly lead to better results. I tested with 10, 25 and 75 categories for both models.

No. Categories	Fully Connected	Convolutional
10	0.6399	0.8999
25	0.4880	0.8880
75	0.4320	0.8799

4.3.1 Discussion

For the FCNN, reducing the number of categories from 75 to 10 resulted in a noticeable improvement in accuracy (from 0.4320 to 0.6399). This aligns with my expectation that fewer categories makes classification easier. Still I am suprised that increasing the classes by 50% only lead to an accuracy reduction of 0.1.

The CNN similarly showed improved performance with fewer categories, reaching an accuracy of 0.8999 with 10 categories. Here the drop in accuracy with increasing categories was even less than the FCNN, indicating that the CNN was better equipped to handle the additional complexity. Interestingly, the CNN's accuracy with 75 categories (0.8799) was still significantly higher than the FCNN's accuracy with 10 categories (0.6399).

5 My Thoughts

Throught my experiments, I've done a thourough exploration of Fully Connected Neural Networks (FCNN) and Convolutional Neural Networks (CNN). I've studied the effects of hyperparameter tuning, data splits, and category sizes on the models performance, and drawn conclusion from these. While there is some randomness in developing and training the models, the results clearly indicate that Convolutional Neural Networks (CNNs) are better suited for image classification tasks compared to Fully Connected Neural Networks (FCNNs).

One noteworthy observation is the significant difference in training times. While running my code on Google Colab I noticed that, on average, the FCNN finished training approximately 2.5 times faster than the CNN with the same parameters and data size. This speed advantage can make FCNNs appealing for scenarios where computational resources or time constraints are critical. However, this comes at the cost of performance. The CNN consistently outperformed the FCNN in terms of classification accuracy and stability, particularly as the number of categories increased.

With my final models, the CNN demonstrated the ability to accurately distinguish between 75 categories, whereas the FCNN struggled to achieve similar accuracy even when limited to classifying just 10 categories. The CNN's performance highlights its robustness in handling more complex classification problems, as its architecture is better equipped to extract and utilize spatial hierarchies present in image data. Still it is more computationally expensive to train, and if training time is more important, FCNN's may be preferrable.